AuBRey KAGAN

DESIGN ENTRY

Software fault tolerance staves off the errors that besiege μP systems

In their struggle with hardware transients and undetected software bugs, designers of μP-based systems have a strong ally in fault-tolerant software.

Unless the software in a microprocessor-based product can tolerate unpredictable hardware transients and software errors, it will not work correctly. No matter how well the hardware system is designed, some electrical noise always gets through to cause software upsets. Also, no matter how well the software has been tested and debugged, software or specification errors can give rise to the same types of problems. These upsets consist of either seemingly random out-of-sequence program jumps or data mutilation. As a result, sometimes the processor simply locks up and all of the outputs freeze in their current state. In other cases, the programs continue to function but with erroneous and possibly dangerous actions. Usually, a manual reset brings the system back into normal operation.

However, the addition of software routines and redundant data structures, with or without minimal hardware enhancements, used to increase a system's fault tolerance. That is, the system can be designed to function to specification even in the presence of software errors and transient hardware failures.

The first step in achieving system fault tolerant is to identify the source of errors and determine how the errors may be detected. The designer can then implement recovery schemes so that the system can continue operating with a minimum of disturbance.

Fault-tolerant systems are usually designed to monitor their own performance continuously and, in the event of a transient error, correct the

Dick Jarrett

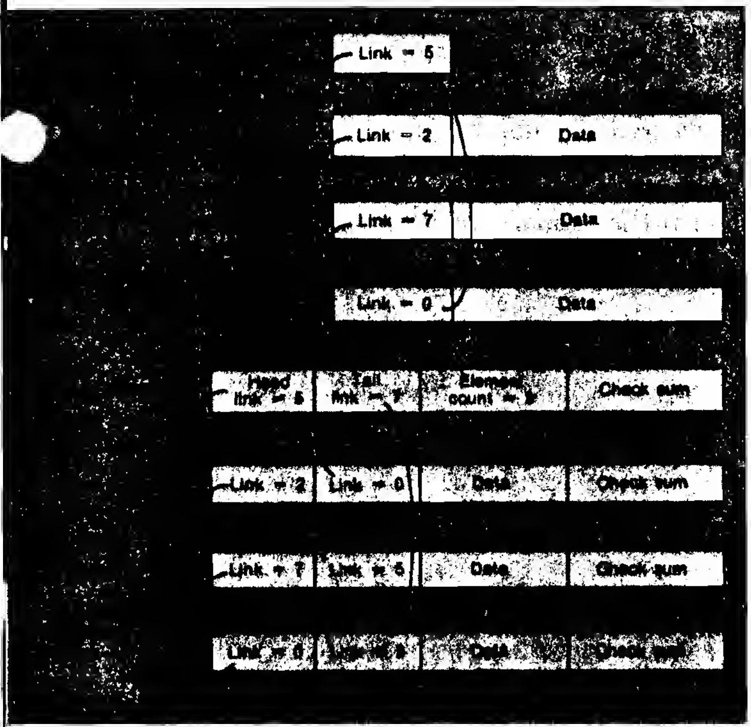
Dick Jarrett is an independent software consultant specializing in the design of microprocessor-based products. Most recently he wrote the software for a fault-tolerant annunciator and recorder. He holds a BSME from MIT and MSME from Northeastern University.

Software Technology: Fault-tolerant software

operation or restart the program. Alternatively, in the case of a permanent (hard) failure, they operate as best they can or allow a backup system to take over.

Such systems should not be confused with so-called fail-safe systems, which shut down to a known safe state after a failure. To do so, any failure must be detected by the software or must automatically lock the system output in a safe state. Unfortunately, the definition of a safe state is not always clear—it is impossible simply to turn off an airplane during takeoff. Fault-tolerant and fail-safe designs in some ways complement and in others contradict one another. Though both need to detect errors, a fault-tolerant system may choose to continue while a fail-safe system would, more conservatively, shut down. Also, some systems are fault-tolerant in the event of a transient error but are designed to enter into a known safe state in the event of a permanent hardware error.

No complex system can successfully handle all possible errors or be considered absolutely fail-safe, but the designer can add software checks and hardware circuits to improve its reliability. Unfortunately, such measures add to a system's complexity, and can therefore instead reduce its reliability. The designer must carefully weigh the many tough tradeoffs involved in making a system as safe and as fault-tolerant as possible.



1. When a singly linked list is corrupted, it can rarely be reconstructed, since the loss of any element invalidates the entire list (a). A doubly linked list's redundancy, coupled with the check sum information (b), greatly facilitates reconstruction.

Quiet, microprocessor zone!

Most of the software upsets in debugged systems are caused by transient hardware problems: any noise on the data, address, or control lines of a microprocessor can lead to, say, an improper instruction fetch. Unfortunately, the sources of noise are legion. One common cause of noise is supply line spikes created by transients on the power line. Since a chip's supply voltage line is connected to every gate and flip-flop in a system, such a spike can potentially change any bit in any data cell. Also, an operator may accidentally discharge a static spark that induces high transient voltages directly in the processor circuitry. Further, radio-frequency interference from walkie-talkies or power drills, say, can also produce noise on internal buses.

Once noise enters a system, it can affect data in RAM, CPU registers, peripheral chip registers, and buses—ranging from a single bit to the entire contents of a random-access memory. If, for instance, the program counter were changed, the processor might execute arbitrary code from RAM or process data bytes as op codes. Such unintended instructions can either alter data or produce an infinite loop. Additionally, if a Halt instruction is encountered, the CPU will stop executing any code.

By entering a valid section of code without first initializing registers and pointers, even a normal program can corrupt its own or another module's data tables. If these tables include the stack, a subroutine could return to a program that did not call it or any other random location. In fact, in machines where the stack is used for both return addresses and data storage, it is doubtful that after jumping to random code the return from the subroutine could be completed successfully. Unfortunately, such unpredictable actions occur in every microprocessor-based system at one time or another.

Not all errors produce such noticeable deviations. Yet it is precisely those errors that go undetected that are the most dangerous. If the set point for a sensor control were accidentally modified, say, the application program might continue working, disrupting the process it controls for hours or even weeks before an operator discovered the problem.

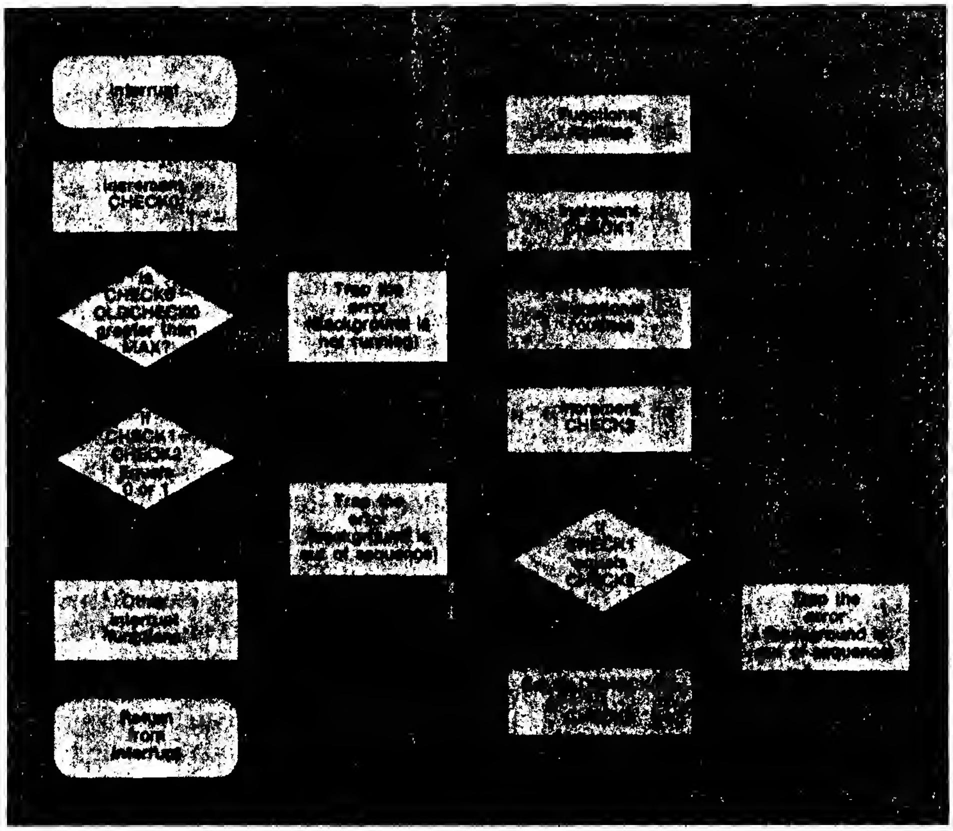
Similar symptoms, different disease

Software errors can create the same problems as hardware transients. An infrequently encountered timing coincidence or an unanticipated external event can cause a program to destroy memory tables or lock into a loop. Indeed, if adequate checks are not carried out, a single erroneous pointer byte can snowball, causing entire sections of memory to be overwritten. Alternatively, a simple error in a linked list can cause a list-searching program to enter an infinite loop.

Many software errors are actually specification errors. Part of a debugged software program, for example, may be put to work in a new application. The functional limitations of the original software are all too easily forgotten—at least until software errors start appearing. Usually, such troubles are the result of inadequate documentation of the original software or of the new specification.

Software engineers must also keep in mind that human beings recognize some situations as impossible while computers do not. Because two objects cannot occupy the same space at the same time, a software designer may not take such a possibility into account. Yet external peripherals can fail or furnish incorrect data, producing physically impossible states that must be included in the decision tables.

Finally, software designers must consider the possibility of permanent hardware damage —often so severe that even the simplest diag-

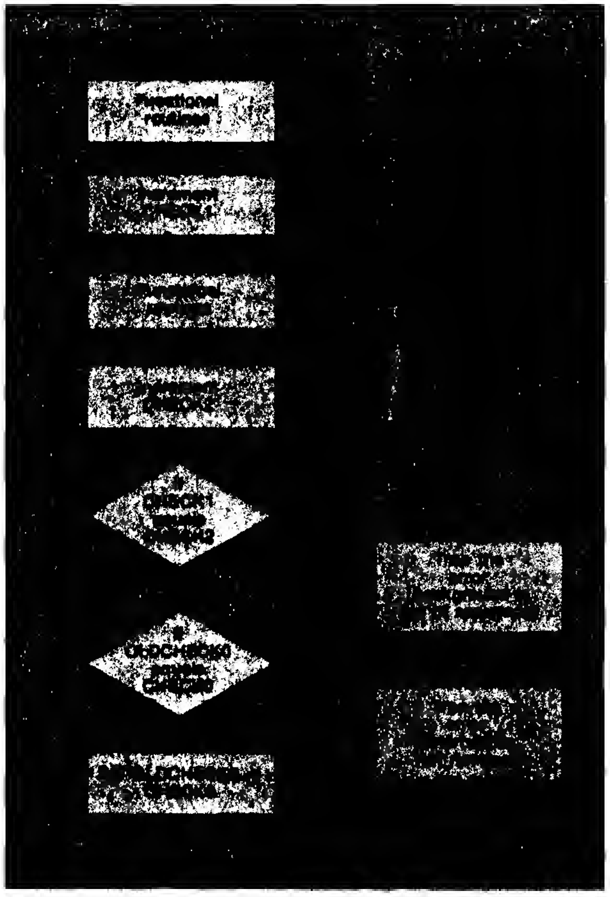

Interrupt
Increment CHECK0
Is CHECK0 – OLDCHECK0 greater than MAX?
Trap the error (background is not running)
CHECK1
CHECK2
0 or 1
Trap the error
Other interrupt functions
Return from interrupt
Functional routines
Increment CHECK1
Increment CHECK2
CHECK2
Trap the error (background out of sequence)

2. In a checkpoint routine (a), any interrupt increments the counter (CHECK0−OLDCHECK0), while the background routine (b) zeroes it. When no background routine runs, the errors are trapped if the counter reaches the value MAX. CHECK1 and CHECK2 ensure that the background routine functions.

nostic routines cannot run. A stuck bit on a control or data line, for example, will prevent the CPU from correctly fetching the op codes of any program. Similarly, a ROM or an address-line error may prevent the CPU from executing an initial test program.

Dealing with the problem

The majority of designers treat such errors as hopeless and attempt a safe shutdown (or allow a redundant system to take over). Hard errors on peripheral chips allow the CPU to operate but not to read inputs reliably or to communicate with other devices. To account for that possibility, the hardware in many systems is designed so that diagnostic routines can test both the inputs and outputs of critical signals. Such loop-back tests, though, are necessarily a part of the application program itself, which must continuously check the signals.



3. An interrupt routine that runs regularly makes sure that the background routine runs each time an interrupt occurs. The counter can be replaced with a range check, which ensures that the background routine has been executed properly.

Obviously, the first step in handling any error is to detect it. Specific tests must thus be included in the software to sound an alarm in the presence of trouble. Error-detecting techniques fall into three broad categories: checks for data consistency, for control flow, and for external hardware.

Associated with each scheme is a "latency of detection," that is, the time elapsing between the actual occurrence of an error and its detection by the hardware or by the software. The sooner an error is uncovered, the less chance it has of doing harm—even though some data errors can lie dormant for extended periods without causing damage.

Still, when the corrupted data is called up, it can cause serious repercussions. Such errors are trapped most quickly by data consistency checks. On the other hand, control flow errors —infinite loops, halts, and out-of-sequence routines—are most easily trapped by internal checkpoint routines and external hardware circuitry.

Data consistency checks are usually designed into the structure of routines as well as into tables. For instance, software should perform some basic checks on the external data that is supplied to the system. Input characters are best examined for syntax; operator responses, for validity; and sensor data, for out-of-range conditions. The routines that take the data from memory usually execute no further checks, but if memory has been altered by a hardware transient or software error, the illegal state cannot be detected. For that reason, the most basic and easiest way to implement a data consistency check is simply to validate data from memory before it is used. Although this procedure does not detect many errors, it does prevent the data from being further damaged by the routine that processes the out-of-range information.

The sum of the parts

More powerful consistency checks can be used on data tables; check sums, for one, can be added to ascertain that the memory has not been modified. At its simplest, a check sum can consist of the arithmetic sum of each of the table entries. Other mathematical schemes such as exclusive-OR check sums and cyclic redundancy checks are also employed. Such techniques are always used for data that is held in magnetic storage, and it is a simple matter to apply them to tables in RAM.

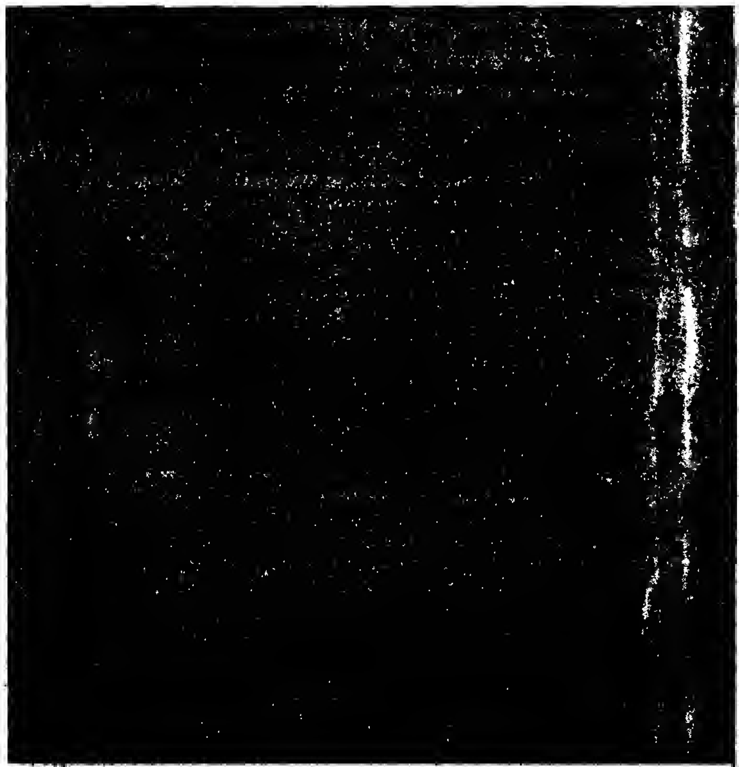
Some data consistency checks are based upon redundant structures. Each entry in a simple linked list consists of a single pointer field and one or more data fields. The pointer, or link, indicates the next member of the list. To find any given entry, a program starts at the top and follows the links until the desired entry is found. To allow data to be found more efficiently, a second, redundant, link can be included as part of the table that usually points to the previous entry. A redundant pointer not only permits backward searches, which in some cases are faster, but allows a diagnostic routine to verify that the list is indeed intact. In general, such an enhanced version of a data structure is more helpful and requires less memory than duplicating the data itself (Fig. 1).

Go with the flow

Software also can be used to check for problems with the control flow, such as determining if a routine is run out of sequence or if a section of code is caught in an infinite loop. A typical failure occurs when low-priority background routines in an interrupt-driven system enter such a loop. The interrupt functions may operate flawlessly, updating their tables and then restoring the system to its original state. Unfortunately, if the background routines are trapped in a loop when called, the illegal state will continue after the interrupt routines restore the system. To guard against that failure, background routines can be made to pass previously defined checkpoints on a regular basis.

Such checkpoints can easily be added to any system. Consider one in which a hardware signal generates an interrupt to periodically update a display. In addition to its normal function, the interrupt routine can be employed to ensure that the background routines are passing the specified checkpoints, which often consist of simple counters. As a checkpoint is passed, the monitored background routine simply increments the counter. The interrupt routine makes sure that the counters are advancing and that they are all consistent. Since the interrupt occurs between checkpoints, the counters can legally be off by no more than one.

What's more, if any routine gets caught in an infinite loop or if processing stops for any reason, the counters cannot advance unless the checkpoint lies within the loop. Multiple checkpoints and a diagnostic routine that verifies—once every loop—whether all of the counters are identical can tell if the interrupt routine is running. A routine that establishes a checkpoint for a random interrupt can be easily implemented (Fig. 2). In much the same way, a routine for a periodic interrupt can be created (Fig. 3). In the example, note that each variable is modified at only one location in the program. Using this philosophy helps guard against software errors, makes nonmaskable interrupts easier to code, and allows routines to read back their own variables for addition checking. Fur-

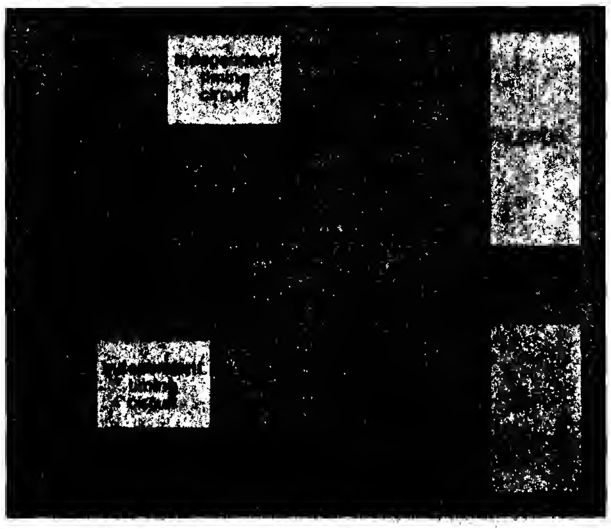


4. To guard against out-of-sequence errors, an 8085 assembly routine first stores its own identification code on the stack. When the routine exits, it checks its ID with the stack to discover if an illegal entry has been made into the routine.

thermore, the use of wraparound integer counters ensures that every possible bit pattern will be written to these RAM locations.

Systems that use an executive program to manage processing tasks can employ a different type of checkpoint to guard against out-of-sequence errors. Whenever a task exits or is suspended, it should inform the executive of its identification code. The executive knows what task should be running and can compare the codes to guarantee that control was not accidentally assumed by some other task that resides in ROM. A stack-based checkpoint scheme either can be implemented for each individual routine or can be incorporated into the executive's logic (Fig. 4).

Equally important are those routines that determine if the stack pointer has been altered. When an out-of-sequence error occurs, the stack pointer is almost always changed inadvertently—say, when data or subroutine return addresses have been placed onto the stack and then never removed. Such errors can be discovered whenever the stack pointer must return to a known location like the highest level of a loop, when the stack must be empty. A simple read of the pointer ensures that all of the data has been removed from the stack. Likewise, if an executive is employed to set up a user stack for a task, the executive can verify that the user has emptied the stack when the task exits.



5. A watchdog timer asserts either a CPU reset or a nonmaskable interrupt to a known valid program if the software-controlled square wave stops (a). Alternatively, the timing circuit imposes a nonmaskable interrupt at regular intervals and a background routine can verify that the watchdog circuit works properly (b).

Finally, since control would be passed to any memory location after an error occurs, designers should place jumps or calls to error-trapping routines in all unused ROM locations. The technique costs practically nothing, since the involved memory is not in use anyway.

Hardware standing sentry

Although the software checks discussed so far are extremely powerful, software alone cannot guard against all errors. Specifically, if a program enters an infinite loop while the interrupts are disabled, an external mechanism is needed to bring the software back to a known state. The hardware used most often in fault-tolerant systems includes some sort of watchdog timer, which comprises an independent timing circuit that periodically tries either to reset the microprocessor or to activate a nonmaskable interrupt.

During normal operation, the software generates a timer reset pulse to the watchdog circuit so that the CPU reset or interrupt never occurs. If, for any reason, the software does not update the timer, the watchdog forces the software into a known state—either reset or interrupted. This method can be used in fail-safe systems as well as fault-tolerant ones. However, if a system depends upon a timer for fail-safe operations, the timer must be allowed to expire periodically so that a failure in the circuit can be detected as well.

A simpler watchdog consists of just a repeating timer tied to a nonmaskable interrupt. In that design, the software has no means of resetting the timer or preventing the interrupt. Rather, the interrupt routine verifies that the checkpoints are being passed through the aforementioned background routines. The advantage of this scheme is that it is self-testing: If the background routines detect that no interrupts are occurring, they can flag the error.

Such a design exemplifies a basic philosophy —a system that cannot execute sophisticated software self-checking should also be incapable of updating the software-driven hardware interlocks that drive the fail-safe outputs. If software checks and hardware interlock updates are interwoven in the programs, they cannot be carried out independently. The second approach, though, cannot be used in systems that depend upon the execution time of individual instructions to generate delays, since an interrupt can occur at any time (Fig. 5).

Note that with either watchdog circuit, the timer must be independent of the CPU. A programmable timer is not acceptable because either the CPU or a hardware transient can permanently disable such a device. Some hardware systems supply a Watchdog Disable bit that is activated only on reset, at which time the watchdog is turned off so that an initialization routine can run. Once the Disable bit is cleared by the software initialization, the software cannot shut down the watchdog. These systems are protected against software errors but not against hardware transients. The same transient that sends the software into an infinite loop can set the bit that disables the watchdog.

Both checkpoints and watchdog circuits have one drawback—the excessive amount of time between the occurrence of an error and its detection. It may be several milliseconds before the watchdog determines that an error has occurred, and in that time, thousands of instructions could have been executed, modifying extensive areas of memory.

Additional hardware checks can shorten the time needed for detection. Dedicated controls, for example, seldom use all the memory that the microprocessor is capable of addressing. In such systems, it is a fairly simple matter to trap, with hardware, any reference to nonexistent memory. Similarly, an attempt to write into ROM can be trapped. Since both of these errors are likely to occur whenever a processor executes random data, the time needed to uncover an error can be reduced dramatically. Hardware parity checks can also be added to trap all hardware-induced single-bit errors in the RAM; however, parity schemes can do nothing when the memory is overwritten by garbled data from an errant software routine.

Once an error has been detected, the software must make a reasonable attempt to restore the system to a legal state. Recovery schemes run the gamut from the simple to the exceedingly complex, and the choice of a particular scheme hinges on the functional characteristics of a given application. Since any error could have been caused by some permanent hardware failure, basic hardware diagnostics should always be carried out. A ROM check sum routine should be invoked to ensure that the program is valid, and a RAM-exercising routine should be executed to ensure that no permanent memory errors exist.

The road to recovery

The simplest recovery scheme consists of resetting the processor and its peripherals, zeroing all of the data tables, and restarting the system. The method may seem crude, but it is adequate in a great number of settings, since transient errors rarely occur more than once every few months. Also, the scheme allows the software to continue running—with some default initialization values. Any operator commands or acquired data are, of course, lost.

Once an error is brought to light, it is desirable to retain as much of the operational data as possible. Simultaneously, the presence of an error throws suspicion on all of the memory, since the exact cause of the problem is unknown. However, the same check sums that are used to detect errors can serve to verify whether individual sections of memory are still intact. Those that are not can be rebuilt through the redundancy designed into the tables. Nevertheless, since error-correcting techniques can corrupt the data even further, the designer must ensure that the probability of missing an error or reconstructing data incorrectly is as small as possible. What's more, the rebuilding routines must make certain that rebuilt tables are consistent with the current state of the system, including any sensors.

Peripheral chips are more difficult to check than RAM, since most manufacturers do not furnish a method of reading back the control registers. Peripheral chips, though, can usually be reinitialized with a minimal loss of operational data. Unfortunately, following a hardware transient, some chips lock into an illegal state and respond only to a hardware reset. To solve that problem, the hardware must be designed so that the peripheral's reset lines are under direct software control.

Software Technology: Fault-tolerant software

Another technique, which combines the two foregoing schemes, is to force a hardware reset whenever an error occurs but not to zero all of the data tables. Data memory comprises two distinct types: scratchpad, and long-term storage. The first is always easier to zero than to rebuild and includes the stack, executive queue tables, communication driver buffers, and various checkpoint flags. Long-term memory contains data that is difficult or impossible to recreate but that can be recovered by error-checking and -correcting logic. Data entered by an operator, information on a controlled process, and time-stamped data are examples of information it is well worth the attempt to save.

Following a hardware reset, an initialization routine can perform the standard hardware diagnostics and zero all of the scratchpad memory. The long-term storage can then be checked for errors and rebuilt. If this memory is deemed hopelessly corrupt, the initialization function can zero these tables or replace them with default data.

Forward recovery

All of the techniques discussed so far belong to the so-called backward error-recovery schemes, which derive their names from the fact that the system backtracks to a known state and essentially restarts. It is also possible to anticipate certain errors and take corrective action based upon the specific fault. Forward recovery, as it is known, is generally used in response to faults external to the basic microprocessor system. For instance, if a printer jams or otherwise malfunctions, the driver software can have a recovery scheme already on hand. These recovery schemes, although important, are actually part of the general application program and are not usually handled the way that trapped errors are.

Finally, in some designs, it is possible just to ignore the problem, force the data into a valid state, and continue processing—say, if new data is continually flushing out old data, as would be the case in a display. The image itself may be somewhat erratic, but it may be more annoying to the viewer to clear the display and start over than to live momentarily with a few erroneous characters.

Similarly, the higher-order bits in certain variables can be ignored. If a variable can legally have only a value between 0 and 15, the upper four bits of the byte variable can always be masked to zero. Following an error, the value may be incorrect, but at least it will not be out of range.

Incorporating fault tolerance into a system is not an absolute science. Rather, the software architect must balance the benefits of specific error detection and correction techniques against the associated increase in both computational overhead and code space.

Most traditional software systems are far too complex to foresee the subtle interactions among all of the routines and variables. Modular programming, in contrast, avoids that pitfall and permits each module to be responsible for its own programs and data tables. In this way, not only does the error recovery scheme become manageable, but changes in the specific implementation of a given module will not affect the recovery scheme of the entire system. Of course, some executive module that oversees all of the various diagnostic, trapping, and recovery functions is usually needed.

Once the basic structure is created, the designer still must decide which variables should be tested and at what frequency. It is impractical to check every variable each time it is used, so some guidelines need to be formulated. For each variable in a module, the programmer should ask two questions: First, what will happen if an incorrect or out-of-range value is stored in the variable? Second, when will that incorrect value be detected or corrected? If no other information will be affected by the one out-of-range variable, then it is often acceptable to postpone its correction.

Point, counterpoint

Nonetheless, the data should be checked at least once every processing cycle or the error may remain undetected forever. If the corruption of a variable could damage other variables within the module, the designer should test the variable before other modifications take place—unless the modified data is easy to rebuild and the error will be detected before serious consequences occur. If an out-of-range variable can wreak havoc with data in other modules, the variable should definitely be checked before it is written to memory.

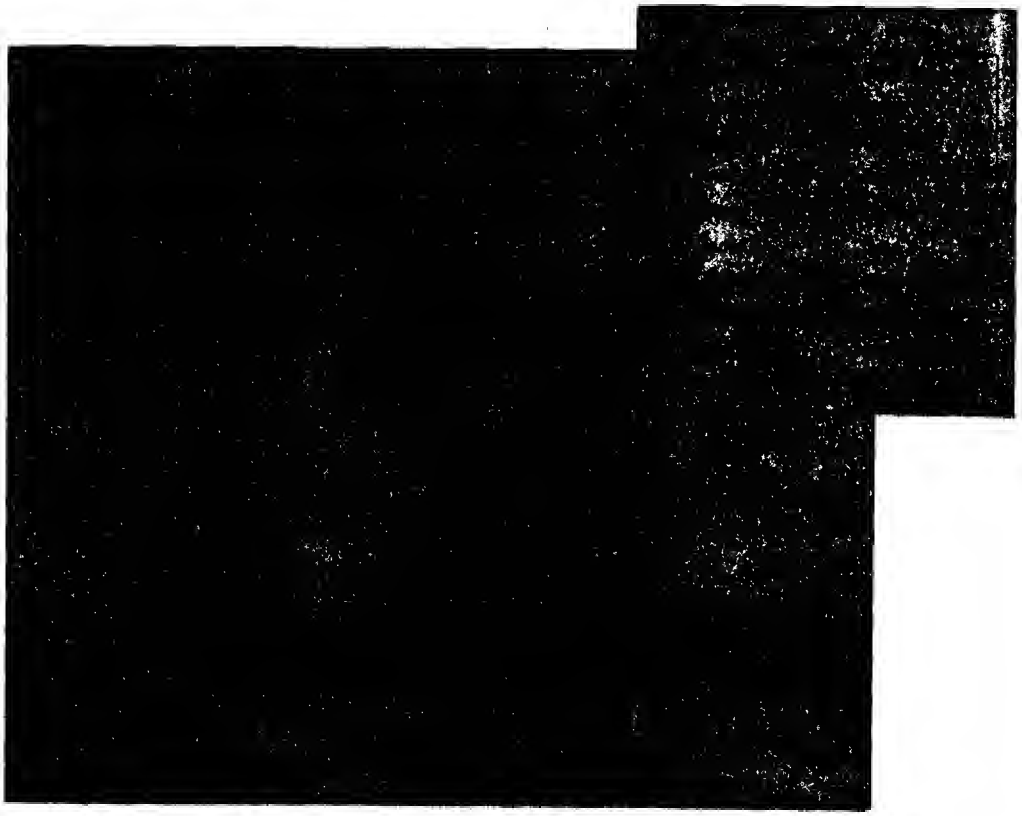
In addition, the functional importance of a module should be taken into consideration. Many more error checks should be included in the module that updates the watchdog timer than in the module that drives a display. Similarly, modules that process data structures that can create infinite loops should receive special attention. Once corrupted, a tree structure or linked list can easily wrap back on itself. A search program that processes such a list should have a maximum depth counter that could be used to terminate a loop.

When considering the need for checks, a designer should keep in mind that rarely altered data has a greater chance of being modified by a hardware transient or software error than freshly changed data. That is particularly true if many other software modules have run since the variable was last examined because they may have corrupted the memory containing the variable. In general, the data stored in machine registers can be deemed safest; recently modified variables can be viewed as intermediately safe; and infrequently modified variables should be the most suspect.

Check, and mate

Check sums for tables are perhaps the most powerful technique available to the designer of fault-tolerant software. Although out-of-range values can be verified readily, incorrect values can only be decided relative to the current state of the machine. By using check sums, a programmer can determine if the value is the same as it was when it was stored. Of course, it must be stored correctly; that is, it should be entered into a table, then summed, and then read back.

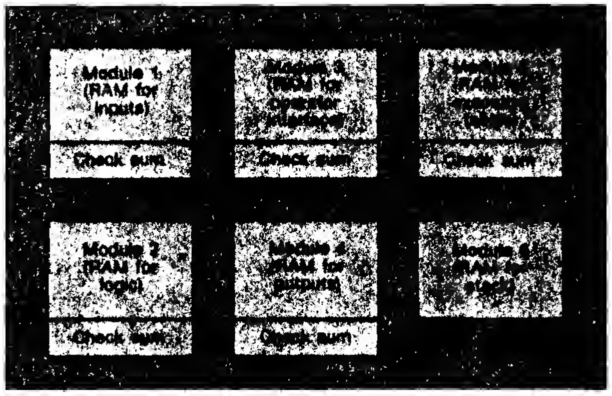
Although static tables are the easiest to check sum, even tables that are being continuously modified may be successfully checked by using a running sum. Rather than recomputing the total each time an entry is changed, the old entry is subtracted from the sum and the new



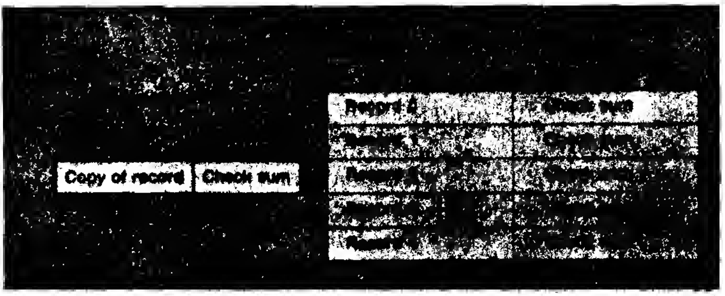
6. In a pseudocode version of the running-check-sum logic (a), a table can be modified at any time (b). The table still retains a valid check sum with the help of the UPDATE procedure.

data is added to it. At any given time, a diagnostic routine can verify that the sum is still correct. If not, then some data within the table has been illegally modified (Fig. 6).

Check sums can also trap design errors in other modules. Once a module is thoroughly debugged and tested, it is incorporated into the system and assumed to be valid. However, new modules may inadvertently and illegally modify the internal variables of an already debugged module. Since the original was thoroughly checked, the illegal modifications can easily be overlooked in the absence of thorough rechecking. Such subtle interactions can lead to bugs that remain undiagnosed for the life of a system. However, if a check sum is run with a table, the error will be trapped by the affected module and bugs can be eliminated during the early stages of testing.



7. To protect memory in a machine with only a single address space, individual RAM modules carry their own check sums. If Module 3, for instance, overflows and modifies Module 4, the check sums permit the error to be trapped and the proper system status to be restored.



8. When updating data tables, it is possible to arrange the sequence of operations to minimize the effect of a system crash by ensuring that the original value remains intact until all interactions with the user have been completed.

Some larger CPUs have address protection mechanisms that isolate programs from one another. Unfortunately, these features are not found in the low-end microprocessors that are put to work in many control applications. Using check sums in these simple machines allows them to approach the same order of protection found in more powerful processors. In fact, a check sum can be used on the larger machines as well, thus achieving a greater degree of protection than it is reasonable to expect from a standard operating system (Fig. 7).

The task of implementing check sums in RAM becomes much easier if larger tables are divided into fixed-length records and if each record is given its own check sum. Moreover, assigning a check sum to every record can make the job of recovering a corrupted linked list or tree far easier. The link field in those records with valid check sums can be considered reasonably secure, and the corrupted records can be thrown out or rebuilt.

Beware the data base

On the other hand, the designer of a database management system would never dream of updating a portion of a record in place. Instead, a copy of the entire record is made. Only after all of the operator interactions are complete and the record is certified as valid is the old record overwritten with new data. The same philosophy can be applied in RAM- or EEPROM-based systems to prevent data from being lost or tables corrupted.

Although a system can crash at any time, only a few microseconds are required to copy a fully checked record from the scratchpad RAM into a permanent RAM table. At the same time, it may take minutes for an operator to decide on the new data to be entered. If the operator works with a copy of the record, rather than the original, the data base's vulnerability is reduced drastically. Moreover, the technique also eliminates logic problems when the operator changes his or her mind and decides to cancel a modification. Finally, if a check sum exists for each record, only one record will be lost in the event of an ill-timed system crash (Fig. 8).

Designers must keep in mind one perverse drawback of fault-tolerant software: the better the job, the smoother the recovery and the less likely it is to even notice an error. Thus a good recovery scheme can—in fact—mask software and hardware errors. In turn, if the designer does not know that a problem exists, it cannot be corrected. However, these errors can be detected with breakpoints and an emulator during the test phase. Once the units are in the field, such errors become even harder to classify and correct since the software automatically compensates for them.

Designers can avoid the dilemma by keeping an error log in RAM that buffers the last few errors (say 16 or so), stored with as much information as is practical. The table should be tested by a check sum (record by record) and should not be cleared during a system reset. If a printer is attached to the product, an appropriate message with error codes may be printed out. If there is no printer, the information can be requested over a serial link or can be recalled using a function button or a sequence on the instrument panel. The error log's data can then be relayed from the field engineer to the software designer and the appropriate action can be taken. Infrequent, or random, errors are probably just that—random hardware transients that result in a valid detection and correction sequence. Repeated errors from the same location, however, are indicative of some undiagnosed hardware or software error.□

Bibliography

Castillo, Xavier, and Daniel P. Siewiorek, "*A Workload Dependent Software Reliability Prediction Model*," Proceedings of the Twelfth Fault Tolerant Computing Symposium, 1982, pp. 279-286.

Glaser, Robert E., and Gerald M. Masson, "*The Containment Set Approach to Crash-Proof Microprocessor and Controller Design*." Proceedings of the Twelfth Fault Tolerant Computing Symposium, 1982, pp. 215-222.

Taylor, David J., and James P. Black, "*Principles of Data Structure Error Correction*," IEEE Transactions on Computers, Vol. C-31, No. 7 July 1982, pp 602-608.

Williamson, Tom, "Designing Microcontroller Systems for Electrically Noisy Environments," Intel Application Note AP-125, February 1982.

Yarkoni, Barry, and John Wharton, "*Designing Reliable Software for Automotive Applications*," SAE Transactions, 790237, July 1979.

How useful?	**Circle**
Immediate design application	**547**
Within the next year	**548**
Not applicable	**549**